Redirecting functions in shared ELF libraries

Written by:
Anthony V. Shoumikhin,
Developer of Driver Development Team,
Apriorit Inc.
http://www.apriorit.com

TABLE OF CONTENTS

1. The problem

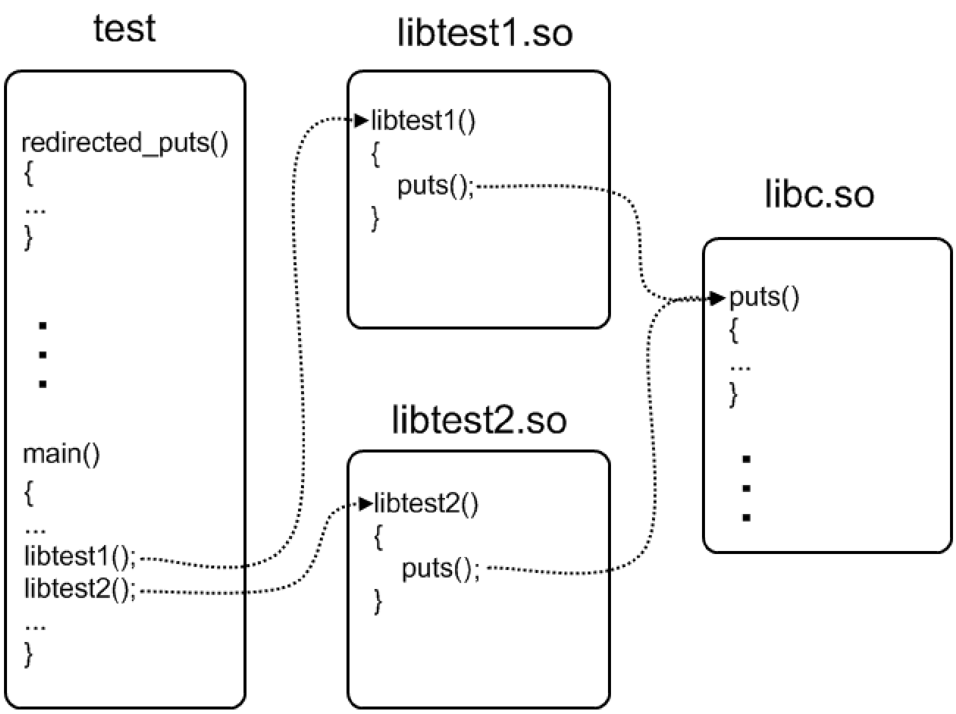
We all use Dynamic Link Libraries (DLL). They have excellent facilities. First, such library loads into the physical address space only once for all processes. Secondly, you can expand the functionality of the program by loading the additional library, which will provide this functionality. And that is without restarting the program. Also a problem of updating is solved. It is possible to define the standard interface for the DLL and to influence the functionality and the quality of the basic program by changing the version of the library. Such methods of the code reusability were called “plug-in architecture”. But let’s move on.

Of course, not every dynamic link library relies only on itself in its implementation, namely, on the computational power of the processor and the memory. Libraries use libraries or just standard libraries. For example, programs in the C\C++ language use standard C\C++ libraries. The latter, besides, are also organized into the dynamic link form (libc.so and libstdc++.so). They are stored in the files of the specific format. My research was held for Linux OS where the main format of dynamic link libraries is ELF (Executable and Linkable Format).

Recently I faced the necessity of intercepting function calls from one library into another - just to process them in such a way. This is called the call redirecting.

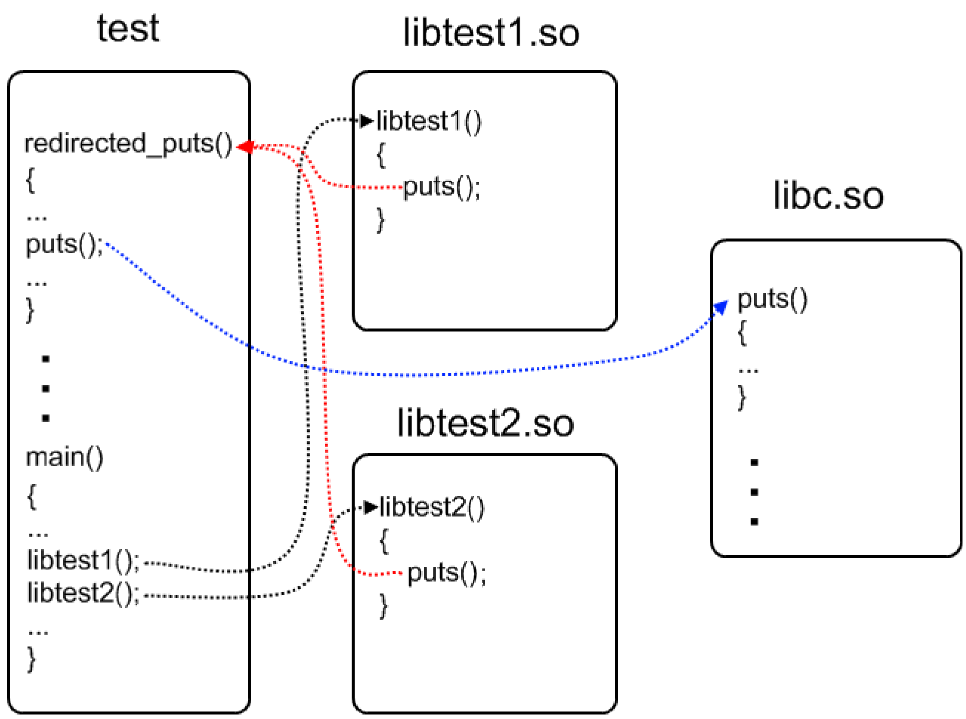
1.1 What does redirecting mean?

First, let’s formulate the problem on the concrete example. Supposing we have a program called «test» on the C language (test.c file) and two split libraries (libtest1.c and libtest2.c files) with permanent contents and which were compiled beforehand. These libraries provide functions: libtest1() and libtest2(), respectively. In their implementation each of them uses the puts() function from the standard library of the C language.

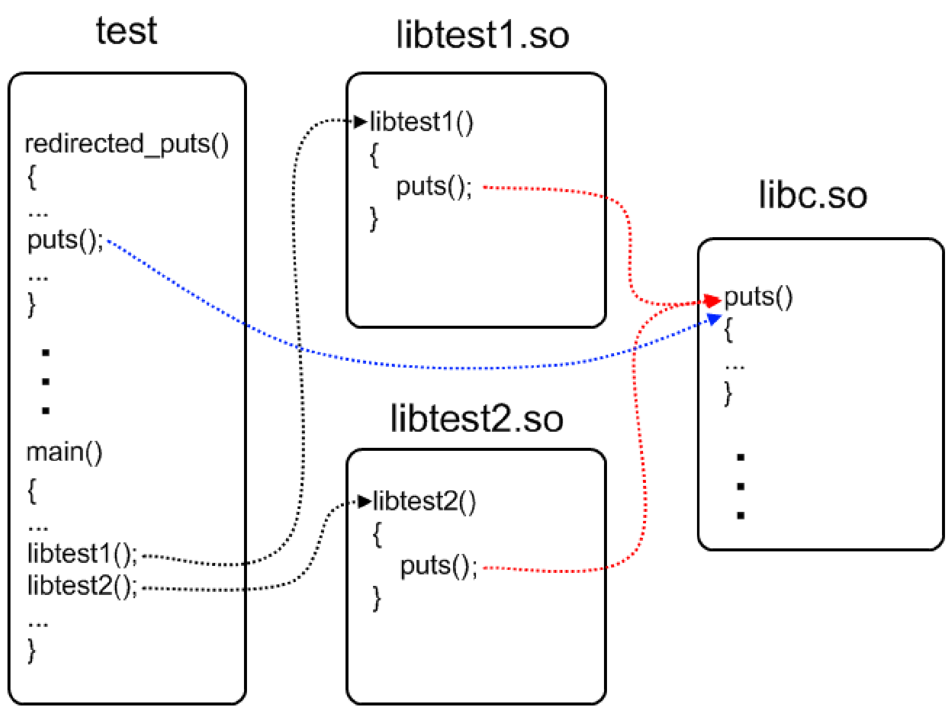


A task consists in the following:

1) To replace the call of the puts() function for both libraries by the call of the redirected puts() function. The latter is implemented in the master program (test.c file) that can in its turn use the original puts() function;



2) To cancel the performed changes, that is to make so that the repeated call of libtest1() and libtest2() leads to the call of the original puts() function.



It is not allowed to change the code or recompile the libraries We can change only the master program.

1.2 Why redirecting?

This example illustrates two interesting specifics of such redirection:

1) It is performed only for one concrete dynamic link library and not for all the process like during the use of LD_PRELOAD environment variable of the dynamic loader. That helps other modules to use the original function trouble-free.

2) It is performed during the program work and does not require its restart.

Where can it be applied? For example, in your program with the variety of plug-ins, you can intercept its calls to system resources or some other libraries. It will not influence other plug-ins and the application itself. Or you can also do the same things from your own plug-in to another application.

How to solve this task? The only variant that came in my mind was to examine ELF and perform corresponding changes in the memory myself.

2. Brief ELF explanation

The best way to understand ELF is to hold your breath and to read its specification attentively several times. Then write a simple program, compile it and examine it in details with the help of the hexadecimal editor, comparing it with the specification. Such method of examination gives the idea of writing some ELF parser because a lot of chore may appear. But do not be in a hurry. Such utilities have been already created. Let's take files from the previous part for the examination:

File test.c

```
#include <stdio.h>
#include <dlfcn.h>

#define LIBTEST1_PATH "libtest1.so"  //position dependent code (for 32 bit only)
#define LIBTEST2_PATH "libtest2.so"  //position independent code

void libtest1();  //from libtest1.so
void libtest2();  //from libtest2.so

int main()
{
    void *handle1 = dlopen(LIBTEST1_PATH, RTLD_LAZY);
    void *handle2 = dlopen(LIBTEST2_PATH, RTLD_LAZY);

    if (NULL == handle1 || NULL == handle2)
        fprintf(stderr, "Failed to open \"%s\" or \"%s\"!\n", LIBTEST1_PATH, LIBTEST2_PATH);

    libtest1();  //calls puts() from libc.so twice
    libtest2();  //calls puts() from libc.so twice
    puts("-----------------------------");

    dlclose(handle1);
    dlclose(handle2);

    return 0;
}
```

File libtest1.c

```
int puts(char const *);

void libtest1()
{
    puts("libtest1: 1st call to the original puts()");
    puts("libtest1: 2nd call to the original puts()");
}
```

File libtest2.c

```
int puts(char const *);

void libtest2()
{
    puts("libtest2: 1st call to the original puts()");
    puts("libtest2: 2nd call to the original puts()");
}
```

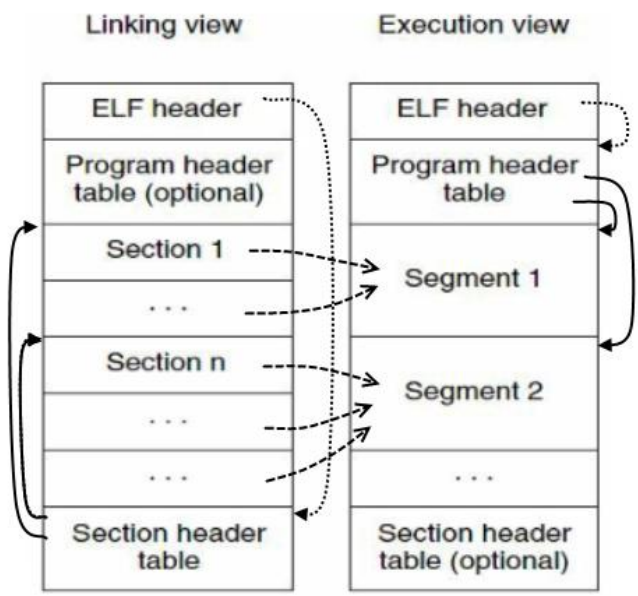
2.1 Which parts does ELF file consist of?

It is necessary to look into such file to answer this question. The following utilities exist for this purpose:

- readelf – a very powerful tool for viewing contents of the ELF file sections
- objdump – it is similar to the previous tool, and it can disassemble the sections
- gdb – it is irreplaceable for debug under Linux OS, especially for viewing places liable to relocation

Relocation is a special term for the place in the ELF file, which refers to the other module symbol. The static (ld) or dynamic (ld-linux.so.2) linker\loader deals with the direct modification of such places.

Any ELF file begins with the special header. Its structure, as well as the description of many other elements of the ELF file, can be found in the /usr/include/linux/elf.h file. The header has a special field, in which the offset from the beginning of the section header table is written. Each element of this table describes some specific section in the ELF file. A section is the smallest indivisible structure element in the ELF file. During loading into the memory, sections are combined into segments. Segments are the smallest indivisible elements of the ELF file, which can be mapped to the memory by the loader (ld-linux.so.2). Segments are described in the table of segments, whose offset is also displayed in the ELF file header.



The most important of them are:

- .text – contains the module code
- .data – initialized variables
- .bss – non-initialized variables
- .symtab – the module symbols: functions and static variables
- .strtab – the names for module symbols
- .rel.text –the relocation for functions (for statically linked modules)
- .rel.data – the relocation for static variables (for statically linked modules)
- .rel.plt – the list of elements in the PLT (Procedure Linkage Table), which are liable to the relocation during the dynamic linking (if PLT is used)
- .rel.dyn – the relocation for dynamically linked functions (if PLT is not used)
- .got – Global Offset Table, contains the information about the offsets of relocated objects
- .debug –the debug information

Let’s perform the following commands for the compilation of files listed above:

```
gcc -g3 -m32 -shared -o libtest1.so libtest1.c
gcc -g3 -m32 -fPIC -shared -o libtest2.so libtest2.c
```

```
gcc -g3 -m32 -L$PWD -o test test.c -ltest1 -ltest2 -ldl
```

The first command creates the dynamic link library libtest1.so. The second creates libtest2.so. Pay attention to the –fPIC key. It makes the compiler generate the so-called Position Independent Code. Details can be found in the next part of the article. The third command creates the executable module with the name “test” by means of the test.c file compilation and by linking it to the already created libtest1.so and libtest2.so libraries. The latter are in the current directory, what is indicated by –L$PWD key. Linking to libdl.so is necessary for using the dlopen() and dlclose() functions.

To start the program, perform the following commands:

```
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
./test
```

That is to add the path to the current directory as a path for the library search to the dynamic linker\loader. The program output will be the next:

```
libtest1: 1st call to the original puts()
libtest1: 2nd call to the original puts()
libtest2: 1st call to the original puts()
libtest2: 2nd call to the original puts()
-----------------------------
```

Now let’s look at the test module sections. Start readelf with the –a key for it. The most interesting examples are displayed below:

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048580
  Start of program headers:          52 (bytes into file)
  Start of section headers:          21256 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
```

This is the standard header of the executable module, a magic sequence in the first 16 bytes. The module type (in this case – executable, but also can be object (.o) and shared (.so)), architecture (i386), recommended entry point, offsets to the headers of segments and sections, and their size are indicated. At the very end of it is the offset in the string table for the headers of the sections.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  ...
  [ 5] .dynsym           DYNSYM          08048200 000200 000110 10   A  6   1  4
  [ 6] .dynstr           STRTAB          08048310 000310 0000df 00   A  0   0  1
  ...
  [ 9] .rel.dyn          REL             08048464 000464 000010 08   A  5   0  4
  [10] .rel.plt          REL             08048474 000474 000040 08   A  5  12  4
  [11] .init             PROGBITS        080484b4 0004b4 000030 00  AX  0   0  4
```

```
  [12] .plt              PROGBITS        080484e4 0004e4 000090 04  AX  0   0  4
  [13] .text             PROGBITS        08048580 000580 0001fc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804877c 00077c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048798 000798 00005c 00   A  0   0  4
  ...
  [20] .dynamic          DYNAMIC         08049f08 000f08 0000e8 08  WA  6   0  4
  [21] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        08049ff4 000ff4 00002c 04  WA  0   0  4
  [23] .data             PROGBITS        0804a020 001020 000008 00  WA  0   0  4
  [24] .bss              NOBITS          0804a028 001028 00000c 00  WA  0   0  4
  ...
  [27] .debug_pubnames   PROGBITS        00000000 0011b8 000040 00      0   0  1
  [28] .debug_info       PROGBITS        00000000 0011f8 0004d9 00      0   0  1
  [29] .debug_abbrev     PROGBITS        00000000 0016d1 000156 00      0   0  1
  [30] .debug_line       PROGBITS        00000000 001827 000309 00      0   0  1
  [31] .debug_frame      PROGBITS        00000000 001b30 00003c 00      0   0  4
  [32] .debug_str        PROGBITS        00000000 001b6c 00024e 01  MS  0   0  1
  ...
  [36] .shstrtab         STRTAB          00000000 0051a8 000160 00      0   0  1
  [37] .symtab           SYMTAB          00000000 005920 000530 10     38  57  4
  [38] .strtab           STRTAB          00000000 005e50 000268 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Here you can see the list of all experimental ELF file sections, their type and mode of loading into the memory (R, W, X and A).

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x007f8 0x007f8 R E 0x1000
  LOAD           0x000ef4 0x08049ef4 0x08049ef4 0x00134 0x00140 RW  0x1000
  DYNAMIC        0x000f08 0x08049f08 0x08049f08 0x000e8 0x000e8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000ef4 0x08049ef4 0x08049ef4 0x0010c 0x0010c R   0x1
```

This is the list of segments, peculiar containers for sections in the memory. Also the path to the special module (dynamic linker\loader) is indicated. It is it to range the contents of this ELF file in the memory.

```
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn
.rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
   07     .ctors .dtors .jcr .dynamic .got
```

And here, the allocation of the sections by segments during the load is displayed.

But the most interesting section, in which information about imported and exported dynamic link functions is stored, is called ".dynsym":

```
Symbol table '.dynsym' contains 17 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND libtest2
     2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     3: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
```

```
     4: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlclose@GLIBC_2.0 (2)
     5: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (3)
     6: 00000000     0 FUNC    GLOBAL DEFAULT  UND libtest1
     7: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlopen@GLIBC_2.1 (4)
     8: 00000000     0 FUNC    GLOBAL DEFAULT  UND fprintf@GLIBC_2.0 (3)
     9: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (3)
    10: 0804a034     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    11: 0804a028     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
    12: 0804879c     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    13: 0804a028     4 OBJECT  GLOBAL DEFAULT   24 stderr@GLIBC_2.0 (3)
    14: 0804a028     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    15: 080484b4     0 FUNC    GLOBAL DEFAULT   11 _init
    16: 0804877c     0 FUNC    GLOBAL DEFAULT   14 _fini
```

Besides other functions that are necessary for the correct program load\roll-out, you can find familiar names: libtest1, libtest2, dlopen, fprintf, puts, dlclose. The FUNC type is meant for all of them and because they are not defined in this module – the index of the section is marked as UND.

The sections ".rel.dyn" and ".rel.plt" are the tables of relocation for those symbols from ".dynsym" that need relocation during the linking in general.

```
Relocation section '.rel.dyn' at offset 0x464 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000206 R_386_GLOB_DAT    00000000   __gmon_start__
0804a028  00000d05 R_386_COPY        0804a028   stderr

Relocation section '.rel.plt' at offset 0x474 contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   libtest2
0804a004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a008  00000407 R_386_JUMP_SLOT   00000000   dlclose
0804a00c  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main
0804a010  00000607 R_386_JUMP_SLOT   00000000   libtest1
0804a014  00000707 R_386_JUMP_SLOT   00000000   dlopen
0804a018  00000807 R_386_JUMP_SLOT   00000000   fprintf
0804a01c  00000907 R_386_JUMP_SLOT   00000000   puts
```

What is the difference between these tables from the point of view of the dynamic link of functions? This is the topic of the next part of the article.

2.2 How do shared ELF libraries link?

The compilation of the libtest1.so and libtest2.so libraries somewhat differed. libtest2.so was compiled with the –fPIC key (to generate Position Independent Code). Let's look how it affected the tables of dynamic symbols for these two models (we use readelf):

```
Symbol table '.dynsym' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     4: 00000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.1.3 (3)
     5: 00002014     0 NOTYPE  GLOBAL DEFAULT  ABS _end
     6: 0000200c     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
     7: 0000043c    32 FUNC    GLOBAL DEFAULT   11 libtest1
     8: 0000200c     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
     9: 0000031c     0 FUNC    GLOBAL DEFAULT    9 _init
    10: 00000498     0 FUNC    GLOBAL DEFAULT   12 _fini
```

```
Symbol table '.dynsym' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

```
     2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     4: 00000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.1.3 (3)
     5: 00002018     0 NOTYPE  GLOBAL DEFAULT  ABS _end
     6: 00002010     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
     7: 00002010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
     8: 00000304     0 FUNC    GLOBAL DEFAULT    9 _init
     9: 0000043c    52 FUNC    GLOBAL DEFAULT   11 libtest2
    10: 000004a8     0 FUNC    GLOBAL DEFAULT   12 _fini
```

So, the tables of dynamic symbols for both libraries differ only in the sequence order of the symbols themselves. It is clear that both of them use undefined puts() function, and grant libtest1() or libtest2(). How have the tables of relocation changed?

```
Relocation section '.rel.dyn' at offset 0x2cc contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000445  00000008 R_386_RELATIVE
00000451  00000008 R_386_RELATIVE
00002008  00000008 R_386_RELATIVE
0000044a  00000302 R_386_PC32        00000000   puts
00000456  00000302 R_386_PC32        00000000   puts
00001fe8  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
00001fec  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
00001ff0  00000406 R_386_GLOB_DAT    00000000   __cxa_finalize

Relocation section '.rel.plt' at offset 0x30c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
00002004  00000407 R_386_JUMP_SLOT   00000000   __cxa_finalize
```

As for libtest1.so, the relocation of the puts() function is found twice in the ".rel.dyn" section. Let's look at these places directly in the module with the help of the disassembler. It is necessary to find the libtest1() function, in which the double call of the puts() function takes place. We use objdump –D:

```
0000043c <libtest1>:
 43c:   55                      push   %ebp
 43d:   89 e5                   mov    %esp,%ebp
 43f:   83 ec 08                sub    $0x8,%esp
 442:   c7 04 24 b4 04 00 00    movl   $0x4b4,(%esp)
 449:   e8 fc ff ff ff          call   44a <libtest1+0xe>
 44e:   c7 04 24 e0 04 00 00    movl   $0x4e0,(%esp)
 455:   e8 fc ff ff ff          call   456 <libtest1+0x1a>
 45a:   c9                      leave
 45b:   c3                      ret
 45c:   90                      nop
 45d:   90                      nop
 45e:   90                      nop
 45f:   90                      nop
```

We have two relative CALL (E8 code) instructions with 0xFFFFFFFC operands. The relative CALL with such operand makes no sense because it directs the control one byte ahead concerning the address of the CALL instruction. If you look at the offset of the relocations for puts() in the ".rel.dyn" section, you can see that they are applied to the operand of the CALL instruction. Thus, in both cases of puts() call, the loader will just rewrite 0xFFFFFFFC so that CALL will jump to the correct address of the puts() function.

The relocation of the R_386_PC32 type works in the described way.

Now let's pay attention to libtest2.so:

```
Relocation section '.rel.dyn' at offset 0x2cc contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000200c  00000008 R_386_RELATIVE
```

```
00001fe8  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
00001fec  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
00001ff0  00000406 R_386_GLOB_DAT    00000000   __cxa_finalize

Relocation section '.rel.plt' at offset 0x2ec contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
00002004  00000307 R_386_JUMP_SLOT   00000000   puts
00002008  00000407 R_386_JUMP_SLOT   00000000   __cxa_finalize
```

The puts() call is mentioned only once and, besides, in the ".rel.plt" section. Let's look at the assembler and perform the debug:

```
0000043c <libtest2>:
 43c:   55                      push   %ebp
 43d:   89 e5                   mov    %esp,%ebp
 43f:   53                      push   %ebx
 440:   83 ec 04                sub    $0x4,%esp
 443:   e8 ef ff ff ff          call   437 <__i686.get_pc_thunk.bx>
 448:   81 c3 ac 1b 00 00       add    $0x1bac,%ebx
 44e:   8d 83 d0 e4 ff ff       lea    -0x1b30(%ebx),%eax
 454:   89 04 24                mov    %eax,(%esp)
 457:   e8 f8 fe ff ff          call   354 <puts@plt>
 45c:   8d 83 fc e4 ff ff       lea    -0x1b04(%ebx),%eax
 462:   89 04 24                mov    %eax,(%esp)
 465:   e8 ea fe ff ff          call   354 <puts@plt>
 46a:   83 c4 04                add    $0x4,%esp
 46d:   5b                      pop    %ebx
 46e:   5d                      pop    %ebp
 46f:   c3                      ret
```

The operands of the CALL instructions are different and intelligent, and this means that they indicate something. It is not a simple padding anymore. Also it is worth mentioning that the recording of 0x1FF4 (0x1BAC + 0x448) into the EBX Registry is performed before the call of the puts() function. The debugger helps to enquiry the initial EBX value, which is equal to 0x448. It means that it will prove useful later. 0x354 address leads us to the very interesting ".plt" section, which is marked as executable as well as ".text". Here it is:

```
Disassembly of section .plt:

00000334 <__gmon_start__@plt-0x10>:
 334:   ff b3 04 00 00 00       pushl  0x4(%ebx)
 33a:   ff a3 08 00 00 00       jmp    *0x8(%ebx)
 340:   00 00                   add    %al,(%eax)
        ...

00000344 <__gmon_start__@plt>:
 344:   ff a3 0c 00 00 00       jmp    *0xc(%ebx)
 34a:   68 00 00 00 00          push   $0x0
 34f:   e9 e0 ff ff ff          jmp    334 <_init+0x30>

00000354 <puts@plt>:
 354:   ff a3 10 00 00 00       jmp    *0x10(%ebx)
 35a:   68 08 00 00 00          push   $0x8
 35f:   e9 d0 ff ff ff          jmp    334 <_init+0x30>

00000364 <__cxa_finalize@plt>:
 364:   ff a3 14 00 00 00       jmp    *0x14(%ebx)
 36a:   68 10 00 00 00          push   $0x10
 36f:   e9 c0 ff ff ff          jmp    334 <_init+0x30>
```

We detect three instructions at the 0x354 address, which we are interested in. In the first of them, the unconditional jump to address indicated by EBX (0x1FF4) plus 0x10 is performed. Having made simple calculations, we get the 0x2004 pointer value. These addresses are in the ".got.plt" section.

```
Disassembly of section .got.plt:
```

```
00001ff4 <.got.plt>:
    1ff4:      20 1f                  and    %bl,(%edi)
       ...
    1ffe:      00 00                  add    %al,(%eax)
    2000:      4a                     dec    %edx
    2001:      03 00                  add    (%eax),%eax
    2003:      00 5a 03               add    %bl,0x3(%edx)
    2006:      00 00                  add    %al,(%eax)
    2008:      6a 03                  push   $0x3
       ...
```

The most interesting thing happens when we dereference this pointer and finally get the unconditional jump address, which is equal to 0x35A. But this is in essence the next instruction! Why should we perform such difficult manipulations and refer to the ".got.plt" section just to jump to the next instruction? What is PLT and GOT at all?

PLT stands for Procedure Linkage Table. It exists in both executables and libraries. It is an array of stubs, one per imported function call.

```
PLT[n+1]: jmp    *GOT[n+3]
          push   #n         @push n as a signal to the resolver
          jmp    PLT[0]
```

A subroutine call to PLT[n+1] will result jumping indirect through GOT[n+3]. When first invoked, GOT[n+3] points back to PLT[n+1] + 6, which is the PUSH\JMP sequence to PLT[0]. Going through the PLT[0], the resolver uses the argument on the stack to determine 'n' and resolves the symbol 'n'. The resolver code then repairs GOT[n+3] to point directly at the target subroutine and finally calls it. And each next call to PLT[n+1], it will be directed to the target subroutine without being resolved by fixed JMP instruction.

The first PLT entry is slightly different, and is used to form a trampoline to the fix up code.

```
PLT[0]: push    &GOT[1]
        jmp     GOT[2]     @points to resolver()
```

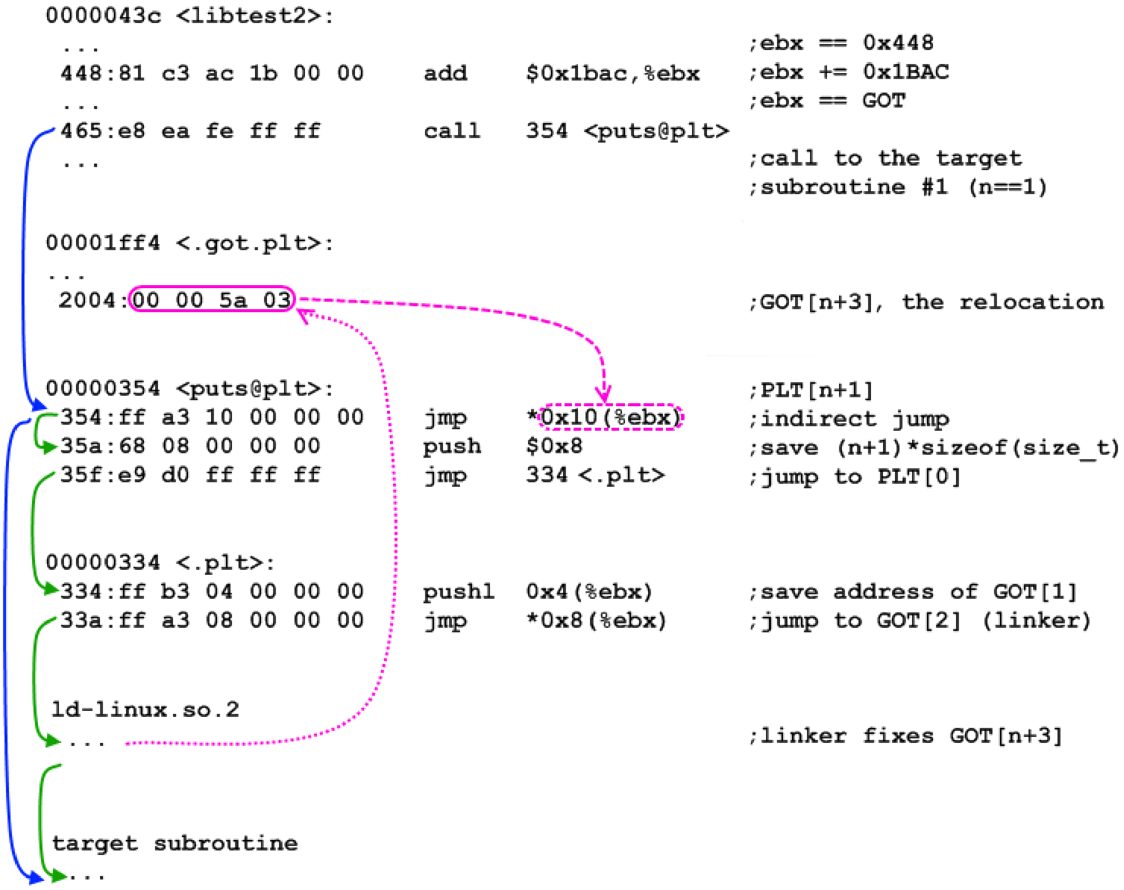
Thread is directed to the resolver routine. 'n' is already in the stack, and address of GOT[1] gets added to the stack. This is the way how the resolver (located in /lib/ld-linux.so.2) can determine, which library is asking for its service.

GOT is the Global Offset Table. The first 3 entries of it are special\reserved. When the GOT is set up for the first time, all the GOT entries relating to PLT fixups are pointing back to the code at PLT[0].

The special entries in the GOT are:

GOT[0]	linked list pointer used by the dynamic loader
GOT[1]	pointer to the relocation table for this module
GOT[2]	pointer to the fixup\resolver code, located in the ld-linux.so.2 library
GOT[3]	
....	indirect function call helpers, one per imported function
GOT[3+M]	
GOT[3+M+1]	
......	indirect pointers to the global data references, one per imported global symbol

Each library and executable gets its own PLT and GOT array.



The relocation of R_386_JUMP_SLOT type, which was used in the libtest2.so library, works in the described way. Other types of relocation refer to the static linking that is why we do not need them.

The difference between the code, which depends on the position of loading to the memory, and the one that does not depend on it (PIC) consists in the methods of allowing of the call of imported functions.

2.3 Some useful conclusions

Let's make some useful conclusions:

- You can get all the information about imported and exported functions in the ".dynsym" section
- If the module was compiled in the PIC mode (-fPIC key), the calls of the imported functions are performed via PLT and GOT; the relocation will be performed only once for each function and will be applied to the first instruction of a specific element in PLT. Information about such relocation can be found in the ".rel.plt" section
- If the –fPIC key was not used during the library compilation, the relocations are performed on the operand of each relative CALL instruction as many times as the calls of some imported function are performed in the code. Information about such relocation can be found in the ".rel.dyn" section

Note: the –fPIC compilation key is required for the 64-bit architecture. It means that the allowing of the calls of imported functions is always performed via PLT\GOT in the 64-bit libraries. Sections with relocations are called ".rela.plt" and ".rela.dyn" on such architecture.

3. The solution

You have to know the following things to perform the redirections of the imported function in some dynamic link library:

1) The path to this library in the file system
2) The virtual address at which it is loaded
3) The name of the function to be replaced
4) The address of the substitute function

Also it is necessary to get the address of the original function in order to perform the backward redirection and thus to return everything on its place.

The prototype of the function for the redirection in the C language is as follows:

```
void *elf_hook(char const *library_filename, void const *library_address, char const *function_name, void
                    const *substitution_address);
```

3.1 What is the algorithm of redirection?

Here is the algorithm of the work of the redirection function:

1) Open the library file.
2) Store the index of the symbol in the ".dynsym" section, whose name corresponds to the name of the required function.
3) Look through the ".rel.plt" section and search for the relocation for the symbol with the specified index.
4) If such symbol is found, save its original address in order to restore it from the function later. Then write the address of the substitute function in the place that was specified in the relocation. This place is calculated as the sum of the address of the load of the library into the memory and the offset in the relocation. That is all. The substitution of the function address is performed. The redirection will be performed every time at the call of this function by the library. Exit the function and restore the address of the original symbol.
5) If such symbol is not found in the ".rel.plt" section, search for it in the "rel.dyn" section likewise. But remember that in the "rel.dyn" section of relocations the symbol with the required index can be found not once. That is why you should not terminate the search loop after the first redirection. But you can store the address of the original symbol at the first coincidence and not to calculate it anymore, it will not change anyway.
6) Restore the address of the original function or just NULL if the function with the required name was not found.

The code of this function in the C language is displayed below:

```
void *elf_hook(char const *module_filename, void const *module_address, char const *name, void const
*substitution)
{
    static size_t pagesize;

    int descriptor;  //file descriptor of shared module

    Elf_Shdr
    *dynsym = NULL,  // ".dynsym" section header
    *rel_plt = NULL,  // ".rel.plt" section header
```

```
    *rel_dyn = NULL;  // ".rel.dyn" section header

    Elf_Sym
    *symbol = NULL;  //symbol table entry for symbol named "name"

    Elf_Rel
    *rel_plt_table = NULL,  //array with ".rel.plt" entries
    *rel_dyn_table = NULL;  //array with ".rel.dyn" entries

    size_t
    i,
    name_index = 0,  //index of symbol named "name" in ".dyn.sym"
    rel_plt_amount = 0,  // amount of ".rel.plt" entries
    rel_dyn_amount = 0,  // amount of ".rel.dyn" entries
    *name_address = NULL;  //address of relocation for symbol named "name"

    void *original = NULL;  //address of the symbol being substituted

    if (NULL == module_address || NULL == name || NULL == substitution)
        return original;

    if (!pagesize)
        pagesize = sysconf(_SC_PAGESIZE);

    descriptor = open(module_filename, O_RDONLY);

    if (descriptor < 0)
        return original;

    if (
        section_by_type(descriptor, SHT_DYNSYM, &dynsym) ||  //get ".dynsym" section
        symbol_by_name(descriptor, dynsym, name, &symbol, &name_index) ||   //actually, we need only the
index of symbol named "name" in the ".dynsym" table
        section_by_name(descriptor, REL_PLT, &rel_plt) ||   //get ".rel.plt" (for 32-bit) or ".rela.plt"
(for 64-bit) section
        section_by_name(descriptor, REL_DYN, &rel_dyn)   //get ".rel.dyn" (for 32-bit) or ".rela.dyn" (for
64-bit) section
       )
    {  //if something went wrong
        free(dynsym);
        free(rel_plt);
        free(rel_dyn);
        free(symbol);
        close(descriptor);

        return original;
    }
//release the data used
    free(dynsym);
    free(symbol);

    rel_plt_table = (Elf_Rel *)(((size_t)module_address) + rel_plt->sh_addr);  //init the ".rel.plt" array
    rel_plt_amount = rel_plt->sh_size / sizeof(Elf_Rel);  //and get its size

    rel_dyn_table = (Elf_Rel *)(((size_t)module_address) + rel_dyn->sh_addr);  //init the ".rel.dyn" array
    rel_dyn_amount = rel_dyn->sh_size / sizeof(Elf_Rel);  //and get its size
//release the data used
    free(rel_plt);
    free(rel_dyn);
//and descriptor
    close(descriptor);
//now we've got ".rel.plt" (needed for PIC) table and ".rel.dyn" (for non-PIC) table and the symbol's index
    for (i = 0; i < rel_plt_amount; ++i)  //lookup the ".rel.plt" table
        if (ELF_R_SYM(rel_plt_table[i].r_info) == name_index)   //if we found the symbol to substitute in
".rel.plt"
        {
            original = (void *)*(size_t *)(((size_t)module_address) + rel_plt_table[i].r_offset);   //save
the original function address
            *(size_t  *)(((size_t)module_address)  +  rel_plt_table[i].r_offset)  =  (size_t)substitution;
//and replace it with the substitutional

            break;  //the target symbol appears in ".rel.plt" only once
        }
```

```
    if (original)
        return original;
//we will get here only with 32-bit non-PIC module
    for (i = 0; i < rel_dyn_amount; ++i)  //lookup the ".rel.dyn" table
        if (ELF_R_SYM(rel_dyn_table[i].r_info) == name_index)   //if we found the symbol to substitute in ".rel.dyn"
        {
            name_address = (size_t *)(((size_t)module_address) + rel_dyn_table[i].r_offset);   //get the relocation address (address of a relative CALL (0xE8) instruction's argument)

            if (!original)
                original = (void *)(*name_address + (size_t)name_address + sizeof(size_t));  //calculate an address of the original function by a relative CALL (0xE8) instruction's argument

            mprotect((void *)(((size_t)name_address) & (((size_t)-1) ^ (pagesize - 1))), pagesize, PROT_READ | PROT_WRITE);  //mark a memory page that contains the relocation as writable

            if (errno)
                return NULL;

            *name_address = (size_t)substitution - (size_t)name_address - sizeof(size_t);   //calculate a new relative CALL (0xE8) instruction's argument for the substitutional function and write it down

            mprotect((void *)(((size_t)name_address) & (((size_t)-1) ^ (pagesize - 1))), pagesize, PROT_READ | PROT_EXEC);  //mark a memory page that contains the relocation back as executable

            if (errno)  //if something went wrong
            {
                *name_address = (size_t)original - (size_t)name_address - sizeof(size_t);  //then restore the original function address

                return NULL;
            }
        }

    return original;
}
```

A full implementation of this function with test examples is attached to this article.

Let's rewrite our test program:

```
#include <stdio.h>
#include <dlfcn.h>

#include "elf_hook.h"

#define LIBTEST1_PATH "libtest1.so"  //position dependent code (for 32 bit only)
#define LIBTEST2_PATH "libtest2.so"  //position independent code

void libtest1();  //from libtest1.so
void libtest2();  //from libtest2.so

int hooked_puts(char const *s)
{
    puts(s);  //calls the original puts() from libc.so because our main executable module called "test" is intact by hook
    puts("is HOOKED!");
}

int main()
{
    void *handle1 = dlopen(LIBTEST1_PATH, RTLD_LAZY);
    void *handle2 = dlopen(LIBTEST2_PATH, RTLD_LAZY);
    void *original1, *original2;

    if (NULL == handle1 || NULL == handle2)
        fprintf(stderr, "Failed to open \"%s\" or \"%s\"!\n", LIBTEST1_PATH, LIBTEST2_PATH);

    libtest1();  //calls puts() from libc.so twice
```

```
    libtest2();  //calls puts() from libc.so twice
    puts("-----------------------------");

    original1 = elf_hook(LIBTEST1_PATH, LIBRARY_ADDRESS_BY_HANDLE(handle1), "puts", hooked_puts);
    original2 = elf_hook(LIBTEST2_PATH, LIBRARY_ADDRESS_BY_HANDLE(handle2), "puts", hooked_puts);

    if (NULL == original1 || NULL == original2)
        fprintf(stderr, "Redirection failed!\n");

    libtest1();  //calls hooked_puts() twice
    libtest2();  //calls hooked_puts() twice
    puts("-----------------------------");

    original1 = elf_hook(LIBTEST1_PATH, LIBRARY_ADDRESS_BY_HANDLE(handle1), "puts", original1);
    original2 = elf_hook(LIBTEST2_PATH, LIBRARY_ADDRESS_BY_HANDLE(handle2), "puts", original2);

    if (NULL == original1 || original1 != original2)  //both pointers should contain hooked_puts() address
now
        fprintf(stderr, "Restoration failed!\n");

    libtest1();  //again calls puts() from libc.so twice
    libtest2();  //again calls puts() from libc.so twice

    dlclose(handle1);
    dlclose(handle2);

    return 0;
}
```

Compile it:

```
gcc -g3 -m32       -shared -o libtest1.so libtest1.c
gcc -g3 -m32 -fPIC -shared -o libtest2.so libtest2.c

gcc -g3 -m32 -L$PWD -o test test.c elf_hook.c -ltest1 -ltest2 -ldl
```

Then start it:

```
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
./test
```

The output will be the following:

```
libtest1: 1st call to the original puts()
libtest1: 2nd call to the original puts()
libtest2: 1st call to the original puts()
libtest2: 2nd call to the original puts()
-----------------------------
libtest1: 1st call to the original puts()
is HOOKED!
libtest1: 2nd call to the original puts()
is HOOKED!
libtest2: 1st call to the original puts()
is HOOKED!
libtest2: 2nd call to the original puts()
is HOOKED!
-----------------------------
libtest1: 1st call to the original puts()
libtest1: 2nd call to the original puts()
libtest2: 1st call to the original puts()
libtest2: 2nd call to the original puts()
```

It indicates the entire fulfillment of the task, which was formulated in the first part of the article.

3.2 How to get the address, which a library has been loaded to?

This interesting question arises during the detailed examination of the function prototype for the redirection. After some research I managed to find out the method of discovering the address of the library loading by its descriptor, which is returned by the dlopen() function. It is performed with the help of such macro:

```
#define LIBRARY_ADDRESS_BY_HANDLE(dlhandle) ((NULL == dlhandle) ? NULL : (void*)*(size_t const*)(dlhandle))
```

3.3 How to write and restore a new function address?

There are no problems with the rewriting of the addresses, which the relocations from the ".rel.plt" section point to. In fact, the operand of the JMP instruction of the corresponding element from the ".plt" section is rewritten. And the operands of such instruction are just addresses.

The situation is more interesting with the applying of relocations to the operands of the relative CALL instructions (E8 code). Their jump addresses are calculated by formula:

address_of_a_function = CALL_argument + address_of_the_next_instruction

Thus, we can find out the address of the original function. Above mentioned formula gives us the value, which has to be written as an argument for the relative CALL in order to perform the call of the necessary function:

CALL_argument = address_of_a_function - address_of_the_next_instruction

The ".rel.dyn" section gets into the segment, which is marked as "R E". It means that you cannot simply write addresses. It is necessary to add the right for record for the page, which the relocation falls to. Do not forget to return everything on its places after the redirection. It is performed with the help of the mprotect() function. The first parameter of this function is the address of the page, which contains the relocation. It must be always multiple of the page size. It is not difficult to calculate it: you should just zero some low bytes of the relocation address (depending on the page size):

```
page_address = (size_t)relocation_address & ( ((size_t) -1) ^ (pagesize - 1) );
```

For example, for pages of 4096 (0x1000) byte size on the 32-bit system, the expression above will be converted to:

```
page_address = (size_t)relocation_address & (0xFFFFFFFF ^ 0xFFF) = (size_t)relocation_address & 0xFFFFF000;
```

The size of one page can be obtained by calling sysconf(_SC_PAGESIZE).

4. Instead of conclusion

As an exercise, you can write a plug-in for Firefox, which will redirect to itself all network calls of, e.g., Adobe Flash plug-in (libflashplayer.so). Thus, you can control all Adobe Flash traffic in the Internet from the Firefox process without the influence on the network calls of the explorer itself and other plug-ins.

Now you have a very convenient tool for the redirection of calls of the imported functions in the ELF dynamic link libraries. Good luck!

Downloads

http://www.apriorit.com/our-experience/articles/9-sd-articles/181-elf-hook

5. Useful links

- http://www.skyfree.org/linux/references/ELF_Format.pdf
- http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- http://vxheavens.com/lib/vsc06.html
- http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html
- http://www.slideshare.net/sanjivmalik/dynamic-linker-presentation
- http://www.codeproject.com/KB/cpp/shared_object_injection_1.aspx
- http://www.linuxjournal.com/article/1060